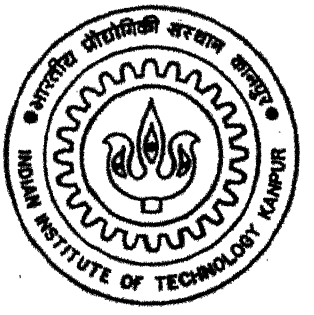
CODE GENERATOR FOR TWINE - RISC

by

MANEPALLI SANKARA SRINIVASA RAO



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JANUARY, 1995

CODE GENERATOR FOR TWINE-RISC

A thesis submitted

in partial fulfillment of the requirements

for the degree of

Master of Technology

by

Manepalli Sankara Srinivasa Rao

to the

Department of Computer Science and Engineering

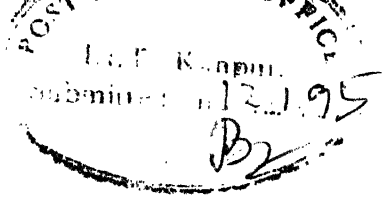
Indian Institute of Technology, Kanpur

January, 1995

Abstract

Twine-RISC is a novel single-chip, processor architecture which exploits the instruction-level temporal parallelism by its well engineered RISC pipeline, and spatial parallelism by allowing multiple threads of computation to co-exist and execute in parallel. This architecture is a hybrid of Von Neumann and Dataflow architectures.

The primary goal of this project is to develop the CODE GENERATOR for Twine RISC. The Code Generator (backend of SISAL Compiler) accepts IF1 (Intermediate Form 1) text file and translates it into Twine RISC code. The source language of the Compiler may be a functional language or a data flow language (SISAL in the present case) which is determined by the frontend of the compiler. The essential components of the backend are the memory allocation (static and the code for dynamic memory allocation) , Register allocation and the Target code genrator, which accepts the required register operands and produces the required target code.



CERTIFICATE

It is certified that the work contained in the thesis entitled *Code Generator for Twine-RISC*, by *M. Sankara Srinivasa Rao* has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Sanjeev Kumar Aggarwal,
Associate Professor,
Department of Computer Science
and Engineering,
IIT Kanpur.

January,1995.

ACKNOWLEDGEMENTS

I am very grateful to my guides, Dr. SANJEEV KUMAR AGGARWAL and Dr RAJAT MOONA, for offering constant guidance throughout my thesis work and to my parents for the support given by them.

I thank pavan, muralidhar, gupta, mahesh, govind, kamal, loc, gaali group and all other friends and classmates who made my stay at IITK enjoyable.

I thank muralidhar, RS, pavan, singhai and srin who were very helpful in the time of need.

I thank my music teachers Mrs Ludimila Chowdary and Santhakka for their guidance and making my stay in IIT a memorable one for my lifetime.

2-1-95 M.Sankar Srinivas Rao

Contents

List of Figures

Chapter 1

Introduction

1.1 High Speed Computing

Recent years have been the advent of many high speed computing systems. These highly parallel machines are a difficult proposition for programmers and compiler writers alike. Programming these systems efficiently is a onerous task. The crux of the problem is in the detection of the inherent parallelism in programs.

The job of the programmer is to transform an abstract problem specification into a concrete implementation. Unfortunately, in conventional programming languages, the problem specification and its implementation are inextricably interwined that it is extremely difficult for the programmer to make a design decision about one without disturbing the other. This problem is even more difficult in a parallel programming environment as the coordination of parallel processes is a very tedious job.

Conventional languages offer little assistance to the programmer in dealing with concurrency. Nor do they help the compiler in the detection of parallelism. In a program, one might encounter many instances where it appears as though two pieces of code are independent; but one cannot be absolutely certain. In all of these cases, the code must be sequenced in order to ensure correctness. In short, the traditional languages are oriented towards sequential processing and are not well-tuned for parallel computing.

An alternative approach for the parallel programming machines is to use concurrent versions of conventional languages. These languages provide special constructs by means of which the programmer can explicitly tell the compiler which parts of a program can safely

execute in parallel. But the parallelism is possible only among the various independent modules; within those modules, code has to be sequenced. Smart compilers may be able to search and find some of the unspecified parallelism, but the amount of the parallelism they can detect is limited. Moreover, such compilers are not widely available.

Programming in conventional languages or parallel versions of the same has stifled many opportunities for parallel execution- what we need is a language with no (or perhaps, few) sequential connotations. Ideally, a parallel programming language must have the following properties :

- It must shield the programmer from such details of the machine as the number of the processors, the topology of the communication network etc.
- Parallelism must be implicit in its semantics.
- It must automatically ensure determinancy.

A single-assignment, applicative language such as SISAL (Streams and Iterations in a Single Assignment Language) [SIS85] seams to be a good choice as a parallel programming language. The following points supports this view.

- Because of the single-assignment property, the language lends itself naturally to parallel execution, except where sequencing is enforced by data dependencies.
- It does not place artificial constraints on the evaluation order.
- It simplifies a part of the work done by the compiler because of the single-assignment rule and the prohibition of aliasing and the side effects in functions, it simplifies the global data flow analysis needed for most of the code optimization technique.

1.2 SISAL Compiler

1.2.1 Structure of SISAL Compiler

Basic structure of a SISAL compiler is shown below. The lexical analysis and the parsing phases are not shown in the figure.

Essentially, the compiler consists of a language specific frontend and a machine specific backend or code generator. The first phase of the compiler is translator from SISAL to

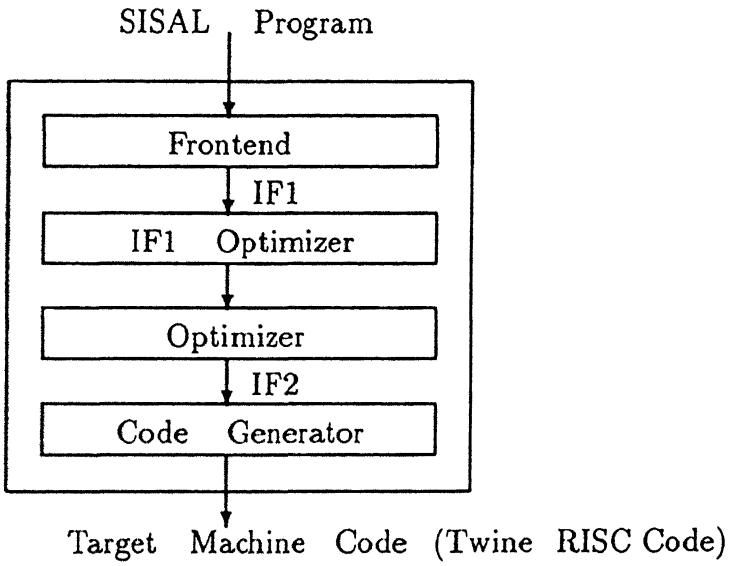


Figure 1.1: Structure of SISAL Compiler

IF1(Intermediate Form 1)[SIS85] .IF1 is a graph language used as an intermediate language in our compiler. The IF1 generator takes a SISAL compilation unit (see next chapter) and translates it into a set of IF1 graphs, one for each SISAL function in the unit. The output of this phase is a text file composed of a sequence of lines, each line representing an "IF1 entity".

The second phase is an optimizer for IF1, most of the classical code optimization techniques can be easily applied to IF1. This part of the compiler could be implemented as soon as the first phase is completed or it could wait till the code generator is also written. Since IF1 cleanly decouples the frontend from the backend, the optimizer module can be easily plugged into the compiler at a later time. In fact, this module may be viewed as a part of the frontend and can be used as a common phase in SISAL compilers for different machines.

The IF1 optimizer is followed by a machine specific optimizer. The output of this optimizer - shown as IF2 in the figure - may be machine dependent. This phase of the compiler is optional. If the compiler contains this phase then the code generator (next phase) should accepts IF2 and produce the target machine code. If this phase is omitted then the code generator accepts IF1.

1.2.2 Previous work done

In the present implementation of the SISAL compiler, the target machine is "Twine RISC" [RS94] . The frontend of the compiler (including the lexical analysis and parsing) has already been implemented[RIY93]. In this version of the compiler the code generator accept IF1 code and produces Twine RISC code. Thus the optimizer phase of the compiler is omitted. The other phase of the compiler (IF1 optimizer) could be implemented later.

1.2.3 Goal of the Project

The primary goal of this project is to develop the CODE GENERATOR for Twine RISC. The Code Generator (backend) accepts IF1 text file and translates it into Twine RISC code. The conversion of Twine RISC code (assemble) to the machine code is done using Assembler, Linker, Loader for Twine RISC. The source language may be a functional or a data-flow language (SISAL in the present case) which is determined by the frontend of the compiler.

The essential components of the backend are the memory allocation (static and the code for the dynamic memory allocation) , Register allocation and the Target code generator, which accepts the required register operands and produces the required target code.

Twine RISC is a processor which combines the advantage of von-Neumann and dataflow architectures. Multiple threads can be efficiently executed in Twine RISC. The multiple RISC pipeline in Twine RISC facilitate simultaneous execution of multiple threads, thus effectively exploiting the instruction level parallelism.

IF1 (Intermediate Form 1) is a hierarchical graph language that decouples the frontend of the compiler from the architecture specific backend. Both SISAL and IF1 were developed by researchers at Lawrence Livermore National Laboratory (LLNL), Digital Equipment Corporation (DEC), Colorado State University (CSU) and the University of Manchester (UM)[SIS85].

1.3 Organization of the thesis

The remainder of this thesis is organized into four chapters. Next chapter gives an overview of the IF1 (Intermediate Form 1). It touches only the relevant features of the language.

In the third chapter, we present the target machine architecture (Twine RISC) and the an overview of the instruction set of Twine RISC.

We describe the main part of the thesis in the fourth chapter. This chapter describes how the code generator has been implemented. It clearly explains how the memory allocation and the register allocation has be performed. It gives the structure and organization of the symbol table and the register file.

In the concluding chapter, we briefly describe the testing done and also look at various possible continuations of this project.

There are three appendices. Appendix A gives the syntax for IF1 and Appendix B gives the instruction set of Twine RISC. Appendix C contains some of the sample programs tested using Code Generator for Twine RISC.

Chapter 2

Intermediate Form1 - IF1

2.1 SISAL Introduction

SISAL (Streams and Iterations in a Single-Assignment Language) [SIS85] is a functional language intented for use on a variety of sequential, vector and data-flow processors. A single assignment language is a language in which the variables are assigned only once and can be used any number of times.

SISAL is designed to express algorithms for execution on computers capable of performing highly concurrent operations. A program in SISAL is a collection of separately translated parts called compilation units. Each compilation unit defines some functions and the nature of the interface that compilation unit is to have with other compilation units. The interface identifies the declared functions that are visible (defined) outside of the current unit and the interface also describes the functions used by this unit that may not be defined there (global). All functions in SISAL must be defined before they can be used. Mutually recursive functions are permitted, in which case one of the functions must be first described in a forward definition.

A function in SISAL computes one or more data values as a function of one or more argument values. Except for invocation of other functions, a function invocation has access only to its arguments; there are no side effects. Further a function retains no state information from one invocation to other; each function invocation is strictly independent. Hence values returned by a function depend only on the argument values presented to it - a SISAL function implements a true function in the mathematical sense.

The data types of SISAL include the basic scalar types boolean, integer, real, double, null and character. Data structure values can be record values, array values, union values and stream values. Each data type has its associated set of operation and predicates. Array, stream, record and union types are treated as mathematical sets of values just as basic data types. The operations for derived types are chosen to support identification of concurrency for execution on a highly parallel processor.

SISAL handles exceptional conditions by producing special error values that serve two purposes.

- An error value indicates that some kind of computational difficulty prevented correct production of a normal answer.
- An error value may preserve portions of structured objects that are known to be correct.

The value *error* is a proper element of every SISAL type. This value is produced in the event of arithmetic or control flow errors.

In SISAL the type of each argument or result value of a function is specified in the function definition's header. The type of each value name used in the body of a function is always directly inferrable from the context in which it is used. The operations in SISAL are designed so that the types of the results can be determined if the types of the operands are known.

Since SISAL is a side-effect free language, subexpressions may be evaluated in any order without effect on computed results.

2.2 Intermediate Language Description

2.2.1 Graphs in IF1

IF1 is a language based on *acyclic* graphs. There are four components to a graph:

- Nodes: Nodes denote operations, such as add or divide.
- Edges: Edges represent values that are passed from node to node.
- Type: Type can be attached to each edge or function.

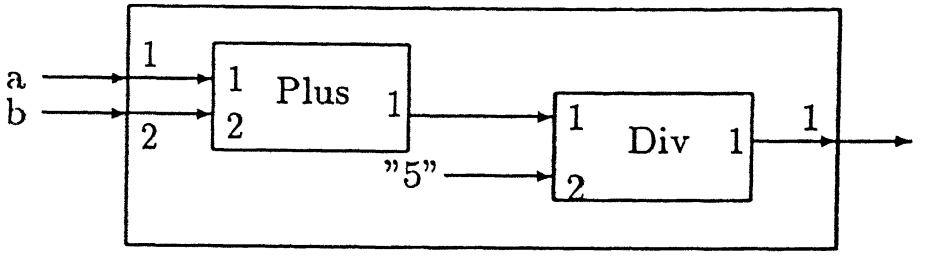


Figure 2.1: Graph of $\frac{(a+b)}{2}$

- Graph: Graph boundaries surround groups of nodes and edges.

For example a graph for the expression (a+b)/5 would be represented by:

The outer box represents the graph boundary. This graph has two ports[SIS85] for input(a and b) and one output port for the result.

The smaller boxes represent nodes. Over fifty nodes are defined in IF1 and a complete list of them is given in Appendix A. Both operations in the example take two input values and return one result, but the number of inputs and the outputs vary according to the operation. For example, the node that builds an array can take any number of inputs, so that array of different sizes can be built. The number of inputs and outputs for each operation node is given in the description of that node. The number inside the node indicate ports, which are used to distinguish multiple inputs and outputs.

The arrows in the example denote edges. Edges represent data paths between nodes(or between nodes and graph boundaries). The edges in the example bring the values for a and b into the graph , and pass their sum from the *Plus* node to the *Div* node, and sent the result out of the graph. The edges also carry type information, which is not shown in the picture.

A special type of edge is used to describe literal constants. Literal edges do not contain a source node or port, but they contain the text of the literal as a string. The "5" in the example is a literal edge. Each edge may carry type information. Strongly typed languages will place type information on the edge in the graph, untyped languages might not give any type information. Types can be specified for user defined types as well as the built_in types.

Comments may be used for any purpose, but they are the appropriate means for giving names to edges, or remembering the line that generated an edge or node.

2.3 The form of IF1

IF1 files comprise a number of lines that contain only printable ASCII characters, and are delimited by newline characters. The first non_blank character on the line distinguishes nodes from edges, etc.

C	Comment
E	Edge
L	Literal edge
N or "{" or "}"	Node
T	Type
G or X or I	Graph

The rest of the line comprises fields that are delimited by white spaces(blanks or tabs) . The number of fields in an IF1 line depends on the type of the line, and in some cases on input to a node. For example, an edge will always have five fields: source node, port on source node, destination node, port on destination node, and type. Any information beyond the expected number of fields is considered to be comment.

Nodes and types contain label fields. IF1 uses integers as labels for nodes and types. The labels for types and nodes need not be disjoint, since we can tell by context whether a reference is to a node or a type. However, the "label scoping" rules for nodes and types differ.

Type labels exist only in one, global scope. Therefore all type labels within a single IF1 file must be unique.

Node labels need only be unique within a given graph. If a graph contains a subgraph, the node label definitions do not extend into the subgraphs. The reason for this difference is that it is important to explicitly import and export values from graphs, while there is no harm in sharing types.

2.3.1 Comments

Comments can appear at the end of any line in an IF1 file ,and as an entire lines that begin with C. Any printable characters (except newline) can appear in a comment.

2.3.2 Nodes

Nodes can be either simple or compound. Simple nodes have the form

```
N    label    Operation
```

where the operation is given by an ASCII string that represents an integers. A list of the operations and their associated integers is given in the BNF in Appendix. Each operation is described in more detail in a later section.

Compound nodes contain subgraphs and span many lines . Their form is:

```
{

.....

} label operation integers(s)
```

Again the operation is denoted by ASCII string that represents an integer. The ellipsis represents an arbitrary number of lines for the subgraph of the compound node. The number of integers at the end of the closing line depends on the number of subgraphs used in the compound nodes. An example of each compound node appears with its description in the later section.

2.3.3 Graph Boundaries

Graph boundaries that denote functions (as opposed to subgraphs of compound nodes) are denoted

```
G  <type reference>  "name"  for a Local function definition

X  <type reference>  "name"  for a Global function definition

I  <type reference>  "name"  for an Imported function
```

The scope of a node labels begin on the line following the G or X and extends until the occurrence of a record beginning with one of the following letters: G,X,I, or an unbalanced }. The label scope does not include the subgraphs of any compound nodes, which appear between balanced "{" and "}". The main purpose of the type field is to specify type of graphs that are functions. Graphs that are subgraphs within compound nodes may choose to not give a type , which is denoted by a zero in the type field.

The graph boundary for imported functions merely associates a type descriptor with a function name. No nodes or edges can follow an imported function descriptor.

The graph boundary serves as a collector of imported and exported values for edges inside it. The boundary is always implicitly labeled "0". When a value is needed from the graph boundary , the edges source node has the label zero. Similarly, when a value is to be returned as the result of a graph, the value is sent to node zero.

Other values are sent and received form nodes numbered beginning at one. Each graph begins numbering its nodes at one and the node labels must be unique within the label scope of a graph.

2.3.4 Edges

Edges represent explicit data dependencies within the graph. They can give information about the type of the value that they represent, and they can also carry extra information in their comment fields. The extra information might be the name associated with the edge, or the source line that generated the edge.

Not all data dependencies are explicitly given in the edge list; some are implicit in the semantics of a compound node and its subgraphs. such implicit connections are described in the semantics of each compound node.

Edges are 6-tuples:(source node, port, destination node, port, reference to type labe, comments) . If no type information is known, the reference should be label zero, meaning "no information provided".

Literals are 5-tuples(destination node, port, reference to type label, string, comment). The string in a literal denotes the value for the literal.

In the example of (a+b)/5 there are five edges:

If we assume that the integer type is "3" we get the following IF1 description of the edge:

```
E    0    1    4    1    3

E    0    2    4    2    3

E    4    1    5    1    3

L              5    2    3    "5"

E    5    1    0    1    3
```

The IF1 code shows that "a" is is imported by the graph on port 1, and that "b" is available on port 2. "5" is a literal that is used at node 5, port 2, and the result (a+b)/5 is exported by the graph on port 1.

In IF1 , arbitrary fan_out of results is possible and is represented by multiple edges with the same source. However, no fan-in is allowed.

2.3.5 Literals

An informal description of different types of literals is given below:

Function Names : are strings of alphanumeric characters. The case of the latters is unimportant.

Boolean Values : are either *True* or *False*

Integer Values : are strings of decimal digits

Single-Precision floating point values : are either strings of decimal digits that contain a decimal point, or exponential notation with an E or e.

Double-precision floating point values : use exponential notation with D or d

Character values : are single printable characters.

Null Value : denoted by *nil*.

Error Values : use text of their representation in higher level language.

Entry	Code	Parameter1	Parameter2
Array	0	base type	
Basic	1	basic	
Field	2	field type	next field
Function	3	argumenttype	result type
Multiple	4	base type	
Record	5	first field	
Stream	6	base type	
Tag	7	tag type	next tag
Tuple	8	type	next in tuple
Union	9	first tag	

Figure 2.2: Type Entries

Type	Code	Type	Code
Boolean	0	integer	3
character	1	null	4
double	2	real	5

Figure 2.3: Basic Type Codes

2.3.6 Type Descriptors

Type descriptors are composed of entries for basic types and entries for type constructors.

Types are organized as a collection of linked lists that tie constructors to base types and functions to arguments and results. Type entries provide type information about edges, whenever the compiler can produce them. Names can be attached to type entries by using same convention as is used for edges. Fields of records can be named in manner shown below.

2.4 Functions

A function is represented by a graph containing the body of the function. In a language that does not import global values(e.g SISAL) the input ports of the graph correspond to the arguments of the function. The results of the graph corresponds to the function results.

The graph receives arguments to the function when it is called (see Call node) and returns a set of values as results. A function name and type are associated with a graph by following a G(graph) record at the outermost level with a type reference and a string(e.g. G 100 sqrt). If the function is to be exported from the current compilation unit the record begins with X , followed by the type reference and a string (e.g. X 100 Boundarycondition).

In order to provide type checking across compilation units, IF1 defines a third form of function header. Lines beginning with I(imports), followed by a type reference and a string , associated types with the names of the imported functions. However no graph body can be present.

Type descriptors for for functions contain two lists: an argument list and a result list. Each of the lists comprise list elements(tuples), which have a type reference and a pointer to the next list element. Pointers refer to type labels (which are integers) and the end of list is denoted by a pointer to *nil*(zero).

2.5 Nodes

Nodes represent operations on values. There are two types of nodes simple and compound. Compound nodes contain subgraphs while simple nodes do not. The semantics of the simple nodes describe the relation of inputs of the nodes to its outputs. The semantics of the compound nodes describe the ways in which the subgraphs of the compound node interact.

Most of the simple nodes have fixed number of input and output ports although there are variable number of inputs to nodes that build structured data types. Compound nodes gather the values needed by their subgraphs, so they generally they have arbitrary number of input and output ports. Any unconnected port is simply unused or "grounded".

The compound nodes are hierarchically defined , so that substructures are denoted by subgraphs.

Simple nodes in an IF1 file are denoted by lines which begin with "N" and are followed by a numeral that represents the node. Lines beginning with "{" and "}" delimit a compound node and its subgraphs. The "}" is followed by a label, an operation code , and an extra information that is specific to that kind of node. A complete list of nodes and their associated numerals is given in Appendix A.

2.5.1 Simple Nodes

Simple nodes are not necessarily uncomplicated nodes, but they have no subnodes. Many of the arithmetic operation nodes are generic. The type of the operation can be determined by examining the types of the input and output arcs. In the following description the below shown notation is followed :

arith	= integer, real, double
algeb	= arith + Boolean
atom	= algeb + char
T	= any type

Below notation is used to construct cartesian products of types

S*T	cartesian product of values of type S and T
$(T)^*$	zero or more occurrences of values of type T
$(T)^+$	one or more occurrences of values of type T
[T]	zero or one occurrence of value of type T

Arithmetic and Boolean Nodes

Some of the arithmetic nodes are :

```
Plus
Times                  algeb * algeb --> ageb
Min
Max
```

The types of both the operands must be same as the type of the result. When the domain is the set of boolean values, Plus and Max accomplish the Or function, while Times and Min accomplish the "And" function.

Exp	arith * arith -->arith

Exp is the function that takes the first input to the power of the second input.

Not	Boolean --> Boolean
Less	
Lessequal	atom * atom --> Boolean
Equal	
Notequal	

Both the operands must of the same type. Notequal and Lessequal represents the Boolean functions exclusive-or and implication.

Iserror	T * T --> Boolean

The first value is a literal whose string denotes one of the error values. Iserror returns true if the second value is the same error value as the first. The special error value "error" matches any error value on the second port.

Miscellaneous Nodes

Some of the miscellaneous nodes are :

Double	real --> double
	integer --> double

If the result is outside the range of double values, the result is an error of type Double.

Floor	real --> integer
	double --> integer

The value returned is the greatest integer less than or equal to the input value. If the result is outside the range of integer values, the result is an error value of type integer.

Noop	$(T)^+$ --> $(T)^+$

Each output is equal to its corresponding input.

Array and Stream manipulation Nodes

Arrays and Streams have nearly same set of operations available on them in IF1. Only the Aadjust, Areplace and Asetl operations are not allowed on streams. The use of the arrays and streams in subgraphs of LoopA, LoopB and Forall may require the use of scatter and gather, which are described in the section on multiple values. Some of the array and stream manipulation nodes are listed below :

Abuild integer * (T)* --> array(T)
integer * (T)* --> stream(T)

Arrays and streams are built by providing a lower bound(ignored for streams) on port one of Abuild, and values on following ports. Any number of values (including zero) may be present, but they must occur on contiguous ports (no gaps). The type of the array or the stream built is found on the edge or edges leaving the Abuild node.

```
Aelement        array(T) * integer --> T
                stream(T) * integer --> T
```

Aelement extracts the element of array or the stream at the index position given on port two. Only one level of subscripting is done.

Acatenate array(T) * $(\text{array}(T))^+$ --> array(T)
stream(T) * $(\text{stream}(T))^+$ --> stream(T)

Acatenate produces the catenation of its input stream or array . A stream will always have a lower bound of one.

Record and Union Manipulation Nodes

Three operation are available on records: create , replace and extract all fields. There are two operations involving objects of type Union: create and distinguish by tag (tagcase) . The latter is a compound node and is discussed in following section.

```
Rbuild          T1 * T2 * ..... * Tn --> record
```

Rbuild creats a record value with n fields. The correspondence between fields and input ports is determined by the ordering of fields in the type description. Port one corresponds to the first field, port two corresponds to second field, etc. The record's type can be obtained by looking at the type of the arc leaving the node.

Rbuild T --> union

The input ports of Rbuild are associated with each tag of the union type. The correspondence between port numbers and tags is determined by the ordering of tags in the type description.

Only one port on an Rbuild node that builds a union can receive a value. The tag is determined by the position of the input port in use. Port one corresponds to first tag, etc.

Relements record --> $(T)^+$

The value for each field of the input record are available at the output ports of Relements. The correspondence between port numbers and the fields is determined by the ordering of fields in the type descriptor, as in Rbuild.

Rreplace record * $(T)^+$ --> record

Rreplace has a port for the input record and one for each field in that record. The first port of the Rreplace receives the record that supplies the basis for the new record. Ports two through (number of fields + 1) give the values to be substituted in the corresponding fields. Any number of ports may receive new values; empty input ports denote fields that are unchanged.

Nodes Dealing with multiple values

An arc with type Multiple(T) represents a sequence of values of type T.

In IF1, there should be a way to transmit all of the loop values int Forall bodies from the ascatter and Rangegenerate nodes. Similarly, to pass sequence of values out of Forall, LoopA, LoopB bodies for the use in test and returns sections. The type constructor Multiple denotes such values and the nodes below operate upon them.

Multiple values have index associated with them. The indices are available for use as values in the loop bodies. They are also used to place an ordering on the values that are

built into streams or arrays(Agather node) and to place values in the correct order for reduction operations.

The below two nodes can only appear in the generator subgraph of forall nodes.

```
Ascatter        array(T) --> multiple(T) * multiple(integer)
                stream(T) --> multiple(T) * multiple(integer)
```

The elements of the input are placed sequentially on port one in the same order as they occur in array or stream. Their corresponding index values are placed on port two.

```
Rangegenerate   integer * integer --> multiple(integer)
```

A sequence of integers is placed on output port one. The (inclusive) range is given on input ports one and two. If the value on port one is greater then the value on port two, the output value is a null multiple.

The rest of the nodes in this section can only appear in the returns subgraph of a Forall, LoopA, or LoopB node.

```
Agather   integer * multiple(T) [* multiple(Boolean)] --> array(T)
          integer * multiple(T) [* multiple(Boolean)] --> stream(T)
```

An array or stream is built form the values on the second port (multiple(T)). Port one gives the lower bound of the array; it is ignored for streams. An edge on port three (multiple(Boolean)) determines whether or not to include the value in the array or stream. A value is included if its corresponding Boolean is true, or if the third port is unused.

```
Reduce
Redleft
Redright   function * T * multiple(T) [* multiple(Boolean)] --> T
Redtree
```

The Reduce nodes take a binary function on port one, with unit value on port two(type T) and applies it to the multiple values on port three (multiple(Boolean)) is present, only those values on port three whose corresponding is Boolean is true will be used in the reduction. Otherwise all the values on port three participate.

The function must be of type(T * T --> T); it is denoted by a literal on port one. The string value of that literal is one of sum, least, greatest The associativity of the operation is either unspecified(Reduce), left (Redleft), right (redright), or pairwise recursive (Redtree).

`Lastvalue` `multiple(T) [ * multiple(Boolean) ] --> T`

`Firstvalue`

Returns the first or the last value of the multiple on port one. If the optional multiple(Boolean) is present on port one , returns the first or last value that occurs when its corresponding boolean value is true.

Firstvalue is only used to provide lower bounds for Agather nodes.

`Allbutlastvalue` `multiple(T) [ * multiple(Boolean) ] --> multiple(T)`

It filters the multiple on port one by removing the last value from the multiple. If the optional multiple(Boolean) value is present on port two, remove the last value of the multiple that occurs when the Boolean is true. If the input multiple is empty, or none of the values in the multiple(boolean) is true, the result of Allbutlastvalue is a multiple containing one error value of type T.

Node Dealing with Function

Call function * $(T)^*$ --> $(T)^+$

The function given on port one is given the arguments (if any) on port two and up. The order of the values is the same as the order of the arguments in the type table entry for the function. The results of the function application are returned on the outputs of the call node.

The function can be any object of type function, but it will be a literal value in VAL and SISAL. The literals type reference field will point to an entry for a function in the type table. The types types of the arguments and the results must match the types of inputs and outputs of the Call node. The name of the function is given in the value string of the literal.

2.5.2 Compound Nodes

Compound Nodes contain subgraphs. The number of subgraphs may be fixed (in Forall, LoopA and LoopB nodes), or may vary (in Select and Tagcase nodes). The semantics of each node relate its inputs and outputs to the inputs and outputs of the subgraphs.

The representation of a compound node in IF1 is :

```
{
G 0  subgraph zero
...
G 0 subgraph one
...


.
.
.


G 0  subgraph n
...
}label OperationCode k a1 a2 a3 .... ak
```

The first line merely marks the beginning of a compound node. Each subgraph begins with a G record (followed by a optional comment) , and terminates when either another G record , or an unmatched } is found. The last line closes the compound nodes and gives a node label, an integer code for the compound node and a list of integers called an association list.

Select

SELECT

$(value)^+$ --> $(value)^+$

N+1 subgraphs (Selector, N alternatives)

The Select node is used to implement a mutiway section. Only a two-way selection exits in SISAL but the Select node could be used to represent "case" construct in other languages.

There are N+1 subnodes.: the Selector and the alternatives. The selector returns exactly one value between 0 and N-1 that determines which subgraph results have to be used.

The association list on the closing line of the compound node identifies the selector and the alternatives. The count of the subgraph must be at least three , since the select needs a selector and at least two alternatives. The first element of the association list identifies the selector subgraph. The following integers give the subgraph number to use for selector values 0,1,...count-2, respectively.

Implicit dependencies

- All inputs ports to the select node are connected to corresponding input ports of all of its subgraphs(class K)
- The result ports of each Alternative subgraphs are similarly connected to the corresponding output ports of the Select Node(class R).

Tagcase

TAGCASE

union * (value)* --> (value)+

K subgraphs (one for each clause in the tagcase)

The Tagcase node is used to access fields of an union object: it must identify a subgraph for each tag in the union type. Each tag will either its own subgraph or several of the tags may share a subgraph.

The association list gives the mapping from tags to subgraphs. The count field will always be equal to the number tags present in the union type. The numbers follow associate a subgraph with a tag, using the ordering of the tags given in the type table entry for the union. The number and types of the results of each subgraph must agree.

Implicit dependencies

- The inputs to the Tagcase node are passed to each subgraph on the class K ports.
- The object selected by the union value of port one of the Tagcase node is passed to each subgraph on port one.

- The results of each subgraph are passed to the outputs of the Tagcase node on the class R nodes.

Forall

FORALL

(value)* --> (value)+

3 subgraphs (Generate, Body, Results)

The Forall node is used to denote independent execution of multiple instances of an expression. It has three subgraphs: the generator, body and results.

The body contains the expression to be evaluated. The generator produces the values for each instance of the body. It does so by producing at least one multiple value on ports in class M . Each element of the multiple value is sent to a distinct instance of the body they are not broadcast to all instances of the body. The values and the instances of the body are ordered and the results subgraph will gather the results in same order. When more than one multiple value is produced by the generator, the first value on each class M edge is sent to the first instance of the body, and so on.

Implicit dependencies

- All inputs to the Forall nodes are available to each subgraph on class K ports.
- The body subgraph receives one value from each class M port.
- The results of the body (class T) are connected to the inputs of the results subgraph.
- The results of the returns subgraph (class R) are connected to the outputs ports of the Forall node.

LoopA

LOOPA

(value)* --> (value)+

4 subgraphs(Initialize, Test, Body, Returns)

LoopA is one of the two iterations construct. It repeatedly the loop body to a set of loop values, stopping and returning the values when the test subgraph returns false. LoopA is a iteration compound node that tests the termination after the body has executed once. It has four subgraphs initialize, test, body, returns. The association list's count contains four, and the four integers that follow will give the number of the subgraph corresponding to Ini, test, Body and returns in that order.

Implicit dependencies

- All the inputs to the LoopA are available to each subgraph on the class K ports.
- The loops values form the output of the initialization subgraph are connected to the inputs of body and returns subgraphs in class L ports.
- The loop values from the output of the body are connected to the inputs of test, body, and returns subgraphs by class L ports.
- The derived loop values from the output of the body are connected to the inputs of the test, body and returns subgraphs by class T ports.
- The results of the returns subgraph are connected to the output ports of the LoopA node by class R ports.
- When the results of the last subgraph is false, the LoopA makes its results available at its output ports.

LoopB

LOOPB

(value)* --> (value)$^+$

4 subgraphs(Initialize, Test, Body, Returns)

LoopB is one of the two iterations construct. It repeatedly the loop body to a set of loop values, stops and returns the values when the test subgraph returns false. LoopA is a iteration compound node that tests the termination immediately after giving the initial values to the loop values. It has four subgraphs initialize, test, body, returns. The association list's count contains four, and the four integers that follow will give the number of the subgraph corresponding to Ini, test, Body and returns in that order.

Implicit dependencies

- All the inputs to the LoopA are available to each subgraph on the class K ports.
- The loops values form the output of the initialization subgraph are connected to the inputs of test, body and returns subgraphs in class L ports.
- The loop values from the output of the body are connected to the inputs of test, body, and returns subgraphs by class L ports.
- The results of the returns subgraph are connected to the output ports of the LoopB node by class R ports.
- When the results of the last subgraph is false, the LoopA makes its results available at its output ports.

Chapter 3

Twine RISC

Twine-RISC[RS94] is a single chip processor architecture which exploits instruction level parallelism by its well engineered RISC pipeline and spatial parallelism by allowing multiple threads of computation to co-exist and execute in parallel. It (Twine-RISC) is a novel design which captures the concept of dataflow and Von Neumann architectures.

3.1 Evolution of Twine-RISC

RISC architectures [PS82] have been derived from the conventional von-Neumann architecture. These architectures are widely used in many of the present day commercial computers. RISC instructions are simple, regular and are usually based on two or three operands. However, in RISC, the inter-dependence of instructions due to the stored program concept of von Neumann computers has been a major bottleneck in the parallel execution of programs.

Dataflow architectures [AC86] offer a possible solution for efficiently exploiting concurrency of computation on a large scale. The computing nodes are "fired" when data arrives and the execution of instructions may not be in the sequence in which they are stored in the memory of a computer [AN89]. However, no existing architecture supports efficient execution of dataflow programs.

Nikhil and Arvind [NA89] proposed *P-RISC*, which combines the ideas of both von-Neumann and dataflow computing. In P-RISC, the program counter(PC), found in von-Neumann computers is eliminated and multiple threads of computation is achieved through the execution of tokens, a concept borrowed from dataflow computing [AC86, AN89]. The

RISC feature of pipelined instruction execution is also effectively utilized. However, in a single processor, multiple threads cannot be simultaneously executed.

Moona, Nandy and Rajaraman [RS94] have proposed a novel architecture called **Twine-RISC** which supports execution of multiple threads in a single processor. *Twine-RISC* has eliminated many drawbacks which are existing in P-RISC and efficiently exploits the fine-grain parallelism which is inherent in most programs.

Dhiren Patel [DH92] had developed a simulator from which he was able to propose some modifications to the original architecture [RS94]. He has also concluded that Twine-RISC is able to fulfill its goals of executing dataflow graphs efficiently with economical architectural frame work.

Dinesh Rao [DIN93] had proposed a simple hardware design for Twine-RISC. In the design proposed by him he had shown two Twine-RISC Streams and had justified it by mentioning the complexities involved as number of Twine-RISC Streams increased.

Shyam Sundar [SS94] developed software support like Assembler, Loader and Simulator for Twine-RISC.

3.2 Twine-RISC architecture

3.2.1 Introduction

In this section, we briefly discuss the processor architecture of *Twine-RISC*. A detailed description of Twine-RISC is available in [DIN93, DH92]

Twine-RISC is a processor which combines the advantages of von-Neumann and dataflow architectures. Multiple threads can be efficiently executed in *Twine-RISC*. The multiple RISC pipelines in *Twine-RISC* facilitate simultaneous execution of multiple threads, thus effectively exploiting the instruction level parallelism. *Twine-RISC* supports *split-phase* transactions between the global memory and the processor through the message processor. All the units of the TRS (described later) operate asynchronously and a handshaking unit is present between each pair of interfaced blocks of a TRS.

Fig. 3.1 illustrates the processor architecture of *Twine-RISC*. In this figure we have shown one Twine-RISC Stream (TRS) completely and the block of second one. There can be any number of such TRSs as permitted by state of art VLSI Technology.

Now we will describe the functionality of different units of Twine-RISC.

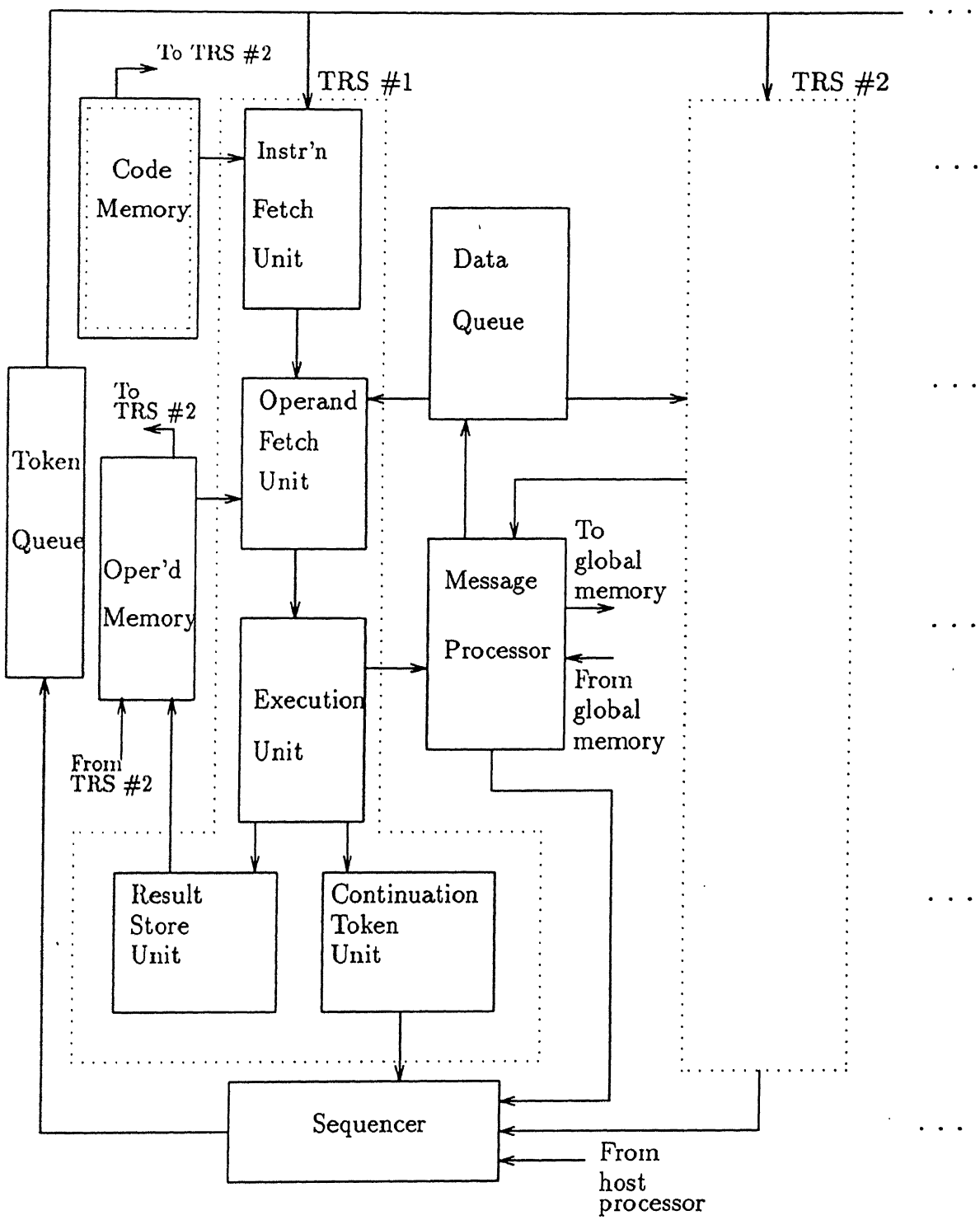


Figure 3.1: Architecture of Twine-RISC

3.2.2 Operand Memory

The Operand Memory (OM) concept in *Twine-RISC* is similar to the register file of conventional RISC processors. It consists of 64 registers of 32-bits each. The OM is shared by all TRSs. The OM is a multi-port memory structure to cater to the demand of multiple TRSs. The read ports are utilized by the OFUs of the TRSs and the write port is used by the RSUs.

3.2.3 Code Memory

The Code Memory (CM) is common to all TRSs and is positioned outside the chip. It holds the instructions and is read-only for the TRSs. A separate host processor is used to initialize the CM.

3.2.4 Token Queue

The continuation tokens for the TRSs are stored in the Token Queue (TQ). A continuation token consists of two pointers, namely the frame pointer (FP) and the instruction pointer (IP). The IP points to the position of the instruction to be executed in the CM and the FP is a base pointer to the data in the OM for a code block. Multiple active invocations of the same code block are possible by the use of frame-relative addressing. A continuation token can be utilized by any TRS. The TQ is initially loaded by the *host processor* through the Sequencer and subsequently, the continuation tokens are supplied by the TRSs. Host can also insert tokens later during the program run to make *Twine-RISC* execute threads asynchronously and thereby handle asynchronous events.

3.2.5 Sequencer

The sequencer serializes the continuation tokens generated by TRSs, MP and Host processor. It sends them one after the other to the TQ. All the tokens present in the TQ are independent of one another and the sequence in which these tokens are stored in the TQ is irrelevant.

3.2.6 Data Queue

The Data queue is similar to TQ and is used to store data coming from the global memory in response to LOAD/LOADX through the Message Processor. These data are written into the DQ and a thread to execute the instruction RESM is added to the TQ. When RESM is executed, data is finally moved from the DQ to the OM and the thread is reinitiated.

3.2.7 Message Processor

MP takes care of the movement of data between the TRSs and the global memory. In case of a read request, the MP receives a response from the global memory controller containing a value, Operand Memory address and continuation token. The MP writes data into DQ and generates a continuation token (0,0) to be stored in the TQ. The MP also directs the LOAD/STORE requests to the global memory. The EXU of the RISC pipeline sends 78 bits data to the MP consisting of 1-bit Read, 32-bit address, 32-bit IP, 6-bit Destination Register (DR) and 1-bit request. The MP receives a similar message (77-bits) from the global memory controller without the request bit and sends it to the DQ.

3.2.8 Instruction Fetch Unit

The Instruction Fetch Unit (IFU) fetches continuation token from TQ and fetches the next instruction from CM (using IP). It also partially decodes the fetched instruction to determine weather the next instruction is to be fetched from the next CM location or not. It also detects the MJOIN instruction and sets the mjoin-lock to enable the atomic execution of MJOIN instruction.

3.2.9 Operand Fetch Unit

This TRS block decodes instructions partially by using three bits of opcode and decides the number of operands to be fetched from the OM. This unit also detects the RESM instruction and if so, fetches operands from the DQ. It then routes the instruction, operands, FP and IP to the EXU.

3.2.10 Execution Unit

It is similar to the ALU of a conventional processor. It also prepares continuation tokens (FP,IP) for branch and other special instructions for thread initiation and forwards it to the Sequencer through the buffer B3. After executing the arithmetic and logical instructions, it sends the result and the address of the destination register to the RSU. Requests for memory read/write are sent to the MP.

3.2.11 Result Store Unit

This is the only stage that can write to the Operand Memory(OM). It writes the value of the result generated by the EXU in the destination register . It also releases the MJOIN *lock* line set by the IFU.

3.3 Instruction Set of Twine-RISC

Twine-RISC has 22 instructions. Each of these instructions can be executed in single clock cycle except one (*MFORK* instruction). These instructions are classified into five major categories viz. Arithmetic and Logic, Copy, Branch, Memory reference and Generation and synchronization of multiple threads. In this appendix, we explain the function of the instructions of *Twine-RISC*. Here we often refer to FP (Frame Pointer) and IP (Instruction Pointer) of TRS [DIN93, DII92]. We have explained these in chapter 4.

A detailed description of how bits are encoded and the actions done by the Twine-RISC is given [DIN93, DII92]. Any register access is starting from FP (Frame Pointer).

3.3.1 Arithmetic and logic group

This group of instructions perform the arithmetic and logic operations.

ADD, SUB, AND, OR, XOR instructions

.

These instructions need three register operands. The operation is done on contents of first two registers and the result is stored in the third register.

The syntax of these instructions is

opcode rs1, rs2, rd where

rs1 - left operand source register.

rs2 - right operand source register.

rd - destination register.

The operation is performed on [FP + rs1] and [FP + rs2]. The resultant value is stored in [FP + rd]. Execution continues from the next location (IP + 1) and no continuation token is generated.

For example: ADD r1, r2, r3

Adds the contents of r1 and r2 and stores the result in r3.

SFTL, SFTR instructions

Shift the operand *left* or *right*. These instructions take one integer (between 0 and 63) and two register operands.

The syntax is

opcode x, rs, rd.

x - number of bits to be shifted.

rs - operand source register.

rd - destination register.

[FP + rs] is shifted to left or right x-bits and the result is stored in [FP + rd]. The execution continues from (IP + 1) and no continuation token is generated.

For example : SFTR 12,r3,r1

Shifts the contents of r3 12-bits to right and stores the result in r1.

3.3.2 MVI instruction

This instruction is to move immediate data (from instruction template) to register.

The syntax is

MVI rd,x

rd - destination register.

x - Integer operand.

Moves x to 12 least significant bits of [FP + rd]. This will not change the 20-most significant bits of [FP + rd]. Absolute value of x should be less than 2047 (0x7ff). The execution continues from (IP + 1) and no continuation token is generated.

For example : MVI r6, 44

Moves 44 into 12-least significant bits of r6.

3.3.3 Branch instructions

These are the jump group of instructions.

JMP instruction

JMP is an unconditional jump instruction (supports direct jump up to a address range of (2^{18}-1).

The syntax is

JMP <label_name>

<label_name> - Label to which the jump has to be made.

The thread will be stopped and a continuation token with IP as address of the location at which label <label_name> is defined and FP as that of parent is generated.

JUMP instruction

JUMP is an unconditional jump instruction (supports jump to a address range of (2^{32}-1)).

The syntax is

JUMP rs.

rs - register specifying the jump offset.

The thread is stopped and a continuation token (FP.IP + [rs]) is generated.

JZ, JP, JPZ, JNZ instruction

These instructions support conditional jump up to 12-bit offset from the current IP. A register operand which contains the value on which the condition is to be tested has to be given.

The syntax for these instructions is

JCND rs, <label_name>

rs - condition operand source register.

<label_name> - Label to which the jump is to be made if the condition is true.

If the condition is true on the value in register rs the current thread is stopped and a continuation token with IP as address of the location at which label <label_name> is

defined) and FP as that of parent is generated. If the condition is false, continuation token is not generated and execution continues from (IP + 1).

3.3.4 Special instructions

These instructions are extensions to the conventional RISC instruction set.

MFORK instruction

MFORK spawns parallel threads of computation from an executing thread. Four possible new threads are organized as 8-bit offsets n1, n2, n3, n4 and grouped into a 32-bit word. This is stored in a register.

The syntax is

MFORK rs, rd

rs - operand source register from which new thread offsets are derived.

rd - destination register.

[FP + rs] is interpreted as four bytes. For each byte, if the value is non-zero, a new thread is generated. A continuation token of (FP,IP + offset value) generated for each non-zero offset value. One continuation token (FP, IP + 1) is always generated. Finally the number of threads generated (including "IP + 1") is stored in [FP + rd]. This value can be used by the MJOIN instruction to synchronize the threads.

MJOIN instruction

The MJOIN instruction helps in the synchronization of multiple threads. This instruction decrements the specified register content by 1 and writes the result back in the same location.

The syntax of MJOIN is

MJOIN rs, rs.

rs - operand source register which contains the number of threads to be synchronized.

MJOIN decrements [FP + r2] by 1 and tests it. If the resultant value is zero, the continuation (FP, IP + 1) is generated else the thread dies.

CHFP instruction

This instruction changes the FP with the first 6-bits of the specified register.

The syntax is CHFP rs.

rs - operand source register.

The 6-least significant bits of rs are loaded to FP. the number in rs should be between 0 and 63 (both inclusive).

3.3.5 Memory based instructions

These instructions are used to move data to and from the global memory.

LOAD instruction

This instruction is used to load data from the global memory to the registers.

The syntax of this instruction is

LOAD rs, rd.

rs - operand source register containing the global memory location.

rd - destination register.

Request is sent to the memory controller for data at address [FP + rs]. The thread dies and is resumed after the memory controller responds. The data returned by the memory is stored in [FP + rd].

LOADX instruction

The syntax is

LOADX a, rd.

a - 6-bit value specifying the address of the global memory location in the instruction.

rd - destination register.

Request is sent to the memory controller for data at address a. The thread dies and is resumed after the memory controller responds. The data returned by the memory is stored in [FP + rd].

RESM instruction

This instruction is executed to move the data from the data queue to the register.

The syntax is

RESM

This instruction stores the data returned by the memory (on load request) in the registers. It reinitiates the thread which stopped on memory request. This instruction should not to be given in assembly program. The system generates it automatically, when necessary.

STORE instruction

This instruction is used to store data into the global memory from the registers.

The syntax is

STORE rd, rs.

rd - operand source register containing the address of the global memory location. (to which the data is to be moved).

rs - operand source register from which data is to be moved to the global memory.

[rs] is sent to memory for storing in [FP + rd]. The thread continues from the next instruction.

STOREX instruction

The syntax is

STOREX x, rs

x - 6-bits specifying the address of the global memory location in the instruction.

rs - operand source register from which data is to be moved to the global memory.

[rs] is sent to memory for storing in x. The thread continues from the next instruction.

3.4 Software Environment for Twine RISC

In this section we briefly discuss about the software environment for Twine RISC developed by [SS94].

3.4.1 Assembler

Assembler takes input from an ASCII file containing the assembly language program for *Twine RISC*, the syntax of which is given in Appendix B. The input is parsed and if it is found to be free of syntax errors then an object file is created (which contains object code suitable for linking). The format of the output file is "Common Object File Format" (COFF) .

3.4.2 Linker

Linker accepts object files and prepares an executable file for the Twine-RISC architecture or another object file suitable for further *Linker* processing (with -r option). The object modules on which link operates are specified on the command line. The Linker input files (object files) are expected to be in "Common Object File Format" (COFF) and the output of the linker (executable file) is also in COFF format.

3.4.3 Simulator

Simulator takes input from an executable file of Twine-RISC machine, produced by the linker. It expects the file in COFF format and simulates the Twine-RISC architecture. The Simulator is written in C language, and runs under Sun OS.

Twine-RISC can execute more than one instruction in every cycle. The number of instructions it can execute is equal to the number of Twine RISC Streams (TRS) on the machine.

Chapter 4

Implementation

The essential components of the backend are the lexical analysis, memory allocation (static and the code for dynamic memory allocation) , Register allocation and the Target code generator, which accepts the required register operands and produces the required target code.

The areas of the implementations where some special attention is to be made are the allocation of memory for derived data types (section 4.5.2) where the size of each element and that of data type can be determined at run time, storage of different types of variables(actual parameters, locals) in the activation record (section 4.6.1), expansion of the compound nodes (section 4.9.2), the organization of the symbol table (section 4.2.1) and the manner in which the exceptional handling (section 4.4) is performed.

4.1 Lexical Analysis

IF1(Intermediate Form 1) [SIS85] has a simple syntax. Looking at the first non-blank character of a line, the type and the number of words in that line can be deduced (excluding the comment). For example if the line starts with a character "E" then it contains 5 operands (source node, port, destination node, port, type) and an optional comment field. If the line begins with a letter "C" then the line contains a comment. In the same manner a line can begin with one of the characters T, L, E, N, G, X, I, {, }, C.

When the preprocessor detects a line beginning with "T" (type descriptor line), it reads the operands, builds a `t_node` (*fields* : `label, para1, para2, para3, next`) and adds this

entry into the type_tab, which is a Hash table for type-entries . The number of buckets in the type_tab is chosen to be 37, as the number of type descriptors vary from 5 to 50.

4.1.1 Storage Structure

If the line in the IF1 file begins with "G", the preprocessing function starts building a new graph. For each new graph encountered a new node called h_node is built (*fields* : side, next, liter, name, type, exp_flag). All the nodes in a graph are stored in a list, whose header is stored in the graph's h_node. For each node in a graph a g_node is created (*fields*: function, number, dependent, depends, next, complex, comp, count_in, count_out). All the edges going out of a node are stored in list whose header is stored in the dependent field of g_node. If the node is compound then the complex field contains the list of sub-graphs of that node and the association list information is contained in the field comp. If the node is simple complex and comp fields of g_node point to *nil*. The node number is stored in the number field of the g_node. The number of in-coming edges is stored in the field depends (excluding literals). The number of in-coming edges (including literals) is stored in the field count_in and the number of out-going nodes is stored in the field count_out. All the literals in a graph are stored in a list , whose header is stored in liter field of h_node of that graph.

4.1.2 Topological Sort

Once the scanning of the input file is completed, the list of nodes in each graph is topologically sorted. Now the graphs are ready for further processing.

4.2 Symbol Table

4.2.1 Symbol Table Organization

The variables in IF1 would be of in the form of tuple (i1,i2) where i1 and i2 are integers. i1 and i2 denote the node and the port numbers respectively. It should be observed that the variables of node would be used only when that particular node is being processed. So the representation chosen for symbol table has a sub-symbol table for each node of the graph. All the variables of that node hash on to the sub-symbol table of that particular node. The number of ports (input + output) for a node can vary from 2 to 40, thus a

hash-table of size 19 is maintained for each sub-symbol table. The *fields* of the sub-symbol table are `num_node`, `buck[19]`, `next`. All sub-symbol tables are stored in the form of a list. The variable can be of type input or output, and the numbering for both input as well as the output ports starts from 1. In order to decrease the clashes between input and output variables, the hashed value of the output variable is complemented with respect to the size of the hash table and then stored in the bucket.

Each entry in the symbol table has the *fields :* `num_port`, `port_type` , `data`, `next`. As the number of entries in each bucket can be more than one, `num_port` and `port_type` fields are used to uniquely identify each entry, `data` field of the `entry` points to the symbol table information of the variable. Here one more level of subscription is used because a variable can have more than one names. (If an edge is present between output port 1 of node 2 and another between input port 5 of node 4, then the name of this variable could be (2,1) or (4,5)).

The data field of the symbol table entry points to a structure of type `st_data` (*fields* : `add`, `data1`, `data2`, `mem_loc`, `type_info`, `usage_count`). The `add` field contains the register number in which the address of the variable is present. If the address is not in a register then this field contains -1. The fields `data1` and `data2` give the register numbers which contain the value of the variable. If the variable is not in register then these fields contains -1. Field `data2` is used for some types of variables such as double, array, stream, multiple, record, union. `mem_loc` gives the position in the activation record, section (section 4.6.1) where the variable is stored, `type_info` gives the type of the variable, `usage_count` gives the number of times the variable would be used in future.

4.2.2 Symbol Table Structure

The structure of symbol-table is given below in detail :

```
struct sym_tab {
     short int num_node;     /*  Node Number    */
     struct st_entry *buck[19];
     struct sym_tab *next;
     };
```

The structure of st_data and sym_table entry is given below :

```
struct st_entry {
     short int num_port;    /* Port Number  */
     short int port_type;   /* Input port or Output port  */
     struct st_data *data;
     struct st_entry *next;  /* Next in the list  */
     };


struct st_data {
     short int add; /* Register in which address is contained */
     short int data1; /* Register in which data1 is contained */
     short int data2; /* Register in which data2 is contained */
     short int mem_loc; /* address in activation record */
     short int type_info;
     short int usage_count;
     } ;
```

4.3 Register File

Twine RISC has a set of 64, 32 bit registers which is called the operand memory. In our implementation of the backend some registers are kept aside for special use. Usage of such registers is described in detail in the following sections.

4.3.1 Structure of Register File

In the implementation, the information about each register is kept in a structure called `reg_type` (*fields* : `node_num, port_num, usage_count, cur_use, comp_flag, add_flag, io_flag`). Fields `node_num, port_num` and `io_flag` are used to identify the variable which is present in the register and inform the symbol table entry for that variable when the register

is deallocated. Field usage_count is used by the *get_reg* function that makes decision about the register to be deallocated, when the need arises. Field cur_use is set if the register is currently being used in the expansion of an IF1 instruction, add_flag specify whether the data or the address of the variable is present in the register, comp_flag is set if the variable is generated in the expansion of a subgraph of a compound node which is also used by the *get_reg* function.

Structure of Reg_file is :

```
struct reg_type {
     short int node_num;
     short int port_num;
     short int usage_count;
     short int cur_use;
     short int comp_flag; /* Node is compound or not  */
     short int add_flag;  /* Address of the variable or data  */
     short int io_flag;  /* Input or Output Variable  */
     };
```

4.3.2 Special Purpose Registers

Stack Pointer

Register #63 : This register is used as the stack pointer for storing the activation records (section 4.6.1) while the program is running. IF1 supports recursive calls of functions. In order to implement recursive calls of functions we need have dynamic memory allocation. As the calls and the returns of the functions takes place in last-in-first-out fashion, we maintain a stack of activation records. The structure of the activation record is given in detail in the next section. Initially this register points to the memory immediately after the program code, whenever there is a function call the value of this register is incremented by the size of the activation record of calling function. Similarly , the same value is subtracted at the time of return from the called function.

Heap Pointer

Register #62: This register is used as the heap pointer to store the values of the variables that are generated while the program is running. When ever memory is to be allocated from the heap, the heap pointer is incremented by the size of the variable. The heap pointer grows in the forward direction.

Register #61: It was found that in the expansion of every IF1 instruction a variable is generated, and the heap pointer need to be incremented by value 4 or its multiple. So register #61 is loaded with 4 and it contains the same value through out the execution od the program.

4.4 Exceptional Handling

Exceptional conditions that occur during the execution of the IF1 program can be detected in the following manner for different data types. If the variable is of simple data type, by comparing the value of the variable with the special value '100..00'(size 32 bits) which is reserved as an *error* value, it can be determined whether any computational difficulty has occurred with that variable. If the variable is of derived data type then by comparing the first word of the variable with the special value '100..00'(size 32 bits), exceptional conditions can be determined.

For the purpose of detecting exceptional conditions IF1 provides an instruction 'iserror'. When this instruction occurs one of the above test is made on the variable depending upon the type of the variable.

4.5 Representation of Data Types and Memory Allocation

4.5.1 Basic Types

Basic data types supported by SISAL are :

Integer

Real

Double

Boolean

Character

Null

The derived types supported by IF1 are :

Record

Array

Stream

Union

Integer

Integer is represented as a 32 bit number. Negative numbers are represented in their 2's complement form. The maximum value of the integer in IF1 is 2^{31}, while the minimum is -2^{31}. Using 2's complement, it's possible to represent $-(2^{31}+1)$ but in our implementation '100..00' is used for detection of over-flow and under-flow.

Real

Real number is a 32 bit floating-point number, its representation is same as that of IEEE 32-bit floating-point representation[CAO88]. The first bit from the left give the sign of the number followed by 8 bit exponent value in excess-127. Remaining 23 bits give the mantissa part of the real number. '100..00' is reserved for representation of error conditions.

Double

Double number is a 64 bit floating-point number, its representation is same as that of IEEE 64-bit floating-point representation. The first bit from the left give the sign of the number followed by 11 bit exponent value in excess-1023. Remaining 52 bits give the mantissa part of the double number. '100..00' is reserved for representation of error conditions.

Character

Character is converted to its corresponding ASCII value, and stored as an integer. Thus a character value ranges from 0 to 127. Error value for character is chosen to be '100..0'.

Boolean

The Boolean value *true* is represented as '00..001' and the *false* is represented as '0...00'. Error value for Boolean type is '10..00'.

4.5.2 Derived Types

The size of the derived data types in IF1 is known only at run-time. Hence the constituents of all the derived types are stored in Heap, and a pointer to this location is maintained by the derived type variable. This would be explained in detail for each derived type of IF1.

Array, Stream, Multiple

If the size of any of the above data types is N, then a memory of size N+3 words is allocated from the heap. The first word of this memory gives the error information of the variable, the next two words give the size and the starting index of the variable, 4th to (N+3)rd word contain the addresses of the elements.

Union

For a Union type of variable a memory of size 3 words is allocated from the heap. First word gives the error information about the variable, the next gives the value of the tag and the last word contains the address of the variable corresponding to the tag value.

Record

If the number of fields of a record type variable is N, then a memory of size (N+2) words is allocated from the heap. The first word of this memory gives the error information of the variable, the next gives the number of fields of the record and the third to the (N+2)nd word gives the addresses of the fields. Third word contains the address of the first field, 4th word contains the address of 2nd field and so on.

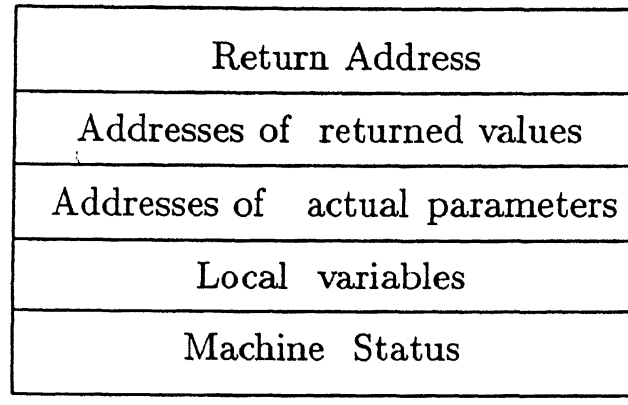


Figure 4.1: Structure of Activation Record

4.6 Runtime System

4.6.1 Activation Record

First word in the activation record contain the address to which the control should be transferred as soon as the execution of the function is completed. For this purpose an unconditional jump instruction is generated at the end of each function.

Jump address is followed by the addresses of the returned values.

Next in the activation record are the addresses of the actual parameters of the function.

Local variable occupy the position after the addresses of actual parameters.

Machine Status field in the activation record contains the values stored in the registers before the call is made. After the called function completes execution, the values stored in this space are loaded back into the respective registers. Only those registers which contain some variables are stored.

4.6.2 Function calls

For each function defined in an IF1 file a structure of type `fun_type` is created. Some of the fields of this structure are `fun_name`, `fun_size` and `type`. Field `fun_name` gives the name of the function and `type` gives the information about the formal parameters and types of the returned values. Field `fun_size` gives the size of the activation record of the function. This field is used when there is a function call. The stack pointer is incremented by the

amount specified by this field when ever a function call occurs and decremented by this amount when the corresponding return.

4.6.3 Function Table

The number of functions defined in a file varies from 2 to 20, so a function table of size 11 is chosen as it is a prime number close to average number of functions defined. The fun_type is hashed on to one of the 11 buckets by applying the a hash_function on function name.

4.7 Macro Table

For every arithmetic operation encountered in the IF1 file, the required operands are obtained and a macro call is made. Whenever a macro call is made, the macro name is stored in the `macro_table` and once the code generation is complete, the code for the macros which are called is produced.

4.8 Register Allocation

4.8.1 Get_reg

This function allocates a register, by deallocating a register if no vacant register is present in the register file. The register allocated must not be a one which is currently being used in the expansion of an IF1 instruction. The register allocated must not be r63, r62 or r61 because they have special use, as mentioned earlier in section 4.3.2

The function starts testing the register file sequentially from register 0 to find a register which is not currently being used and whose `usage_count` is zero. Upon finding such register the function returns that register number, else if it comes to the end of the register file then the following test is made.

In this test, the function tries to find out a register which is not being used currently and whose `add_flag` is set, i.e. register which contains address of a variable. From the set of registers that satisfy the above conditions, the one which has least `usage_count` is selected. The function should also inform the symbol table that this particular variable is removed from the register and the register number is returned.

If the above test fails then the functions tries to find out a register satisfying the conditions : The register is not being used currently and the register contains the value of some variable i.e. `add_flag` is 0. From the set of registers satisfying these conditions, the register with least usage_count is chosen and the symbol table is informed about the deallocation and the register number is returned by the register allocation routine.

4.9 Target Code Generator

For each of the function defined in the IF1 file, a new symbol table and a new register file is built. The symbol table entries are made for the input parameters and the returned values. The space occupied by the input and the returned values in the activation record is shown in the activation record description. All the nodes in the function are sorted topologically by the preprocessing routine. Then each of these nodes are passed on to code_generation routine, in the order specified by the preprocessing routine.

The code_generation routine evaluates the usage_count of the output variables of the node being processed, makes an entry for each of this variable in the corresponding symbol table and stores the value of the usage_count in the symbol table. Now the routine checks whether all the input variables are in the registers, if any of the input variable is not in the register, then it calls the get_reg() routine, obtains a register and stores the required variable in the register. Once all the input variables are in the registers, it tries to get the intermediate registers, used in the evaluation of the node being processed.

4.9.1 Simple Node Expansion

The code_generation routine tries to find out the type of the node. If the node is of type arithmetic then a macro_call is made with the registers as arguments. In case the node is not compound node, the code for IF1 instruction is written into the output file. If the node, is a "Call" node, then the stack pointer is incremented by the size of the activation record of the current function being processed and unconditional jump is made to the called function. As the size of the activation record of the calling function would be known only after processing the function, an entry is left for this size and filled up later when back-patching is performed. After the unconditional jump, the value of the stack pointer is restored.

4.9.2 Compound Node Expansion

If the node being processed is a compound node then the sub-graphs of the node are expanded by the implicit dependency rules of that compound node. For example, if the compound node is "Select", then the selector sub-graph is expanded before any of the alternative sub-graphs. The flow of variables between various sub-graphs and between sub-graphs and the main graph, is given in detail, for each of the compound node in the chapter 2.

As soon as the code for the IF1 instruction is produced, the usage_count of the input variables is decremented by one. This change in the usage_count is stored in both symbol table and the register file. The register numbers in which the outputs produced by the IF1 instruction are also stored in both symbol table and the register file. Now the out-going edges of the node are processed, by making the symbol table entries for the ports of the nodes that are connected to the out-going edges of the node processed.

Chapter 5

Conclusions

5.1 Testing

The implemented Code Generator for Twine RISC was tested for around 20 IF1 programs. Some of the test cases and the generated Twine RISC code are shown in Appendix C. The generated Twine RISC code is tested using the *simulator* for Twine RISC and was found to be correct. Initially the generated code contained some errors which were corrected.

The test programs are chosen in such a manner that all types of nodes in IF1 are covered at least once. The size of the programs tested varies from 20 lines to 50 lines. The programs tested are taken from the manual for SISAL and IF1[SIS85]. The Twine RISC code generated for even a small IF1 program contained more then 100 RISC instructions due to compactness of IF1. If the test case contains a conditional statement then the tests are made for different sets of input values so that every instruction in the program is covered one or more number of times.

5.2 Justification and improvement of IF1

The research on IF1 started as an attempt to design an intermediate language for a higher-level data flow language (SISAL). IF1 has changed form dramatically since its first draft. The major change occurred at a meeting held at DEC in Hudson, in 1983 [SIS85] . At this time the hierarchies of graphs were introduced, eliminating the need for explicit control lines and nodes. Since then, the need for uniquely numbered nodes has been eliminated

and extra type information is now placed in graphs to ensure type correctness when linking functions from different compilation units.

IF1 has not only been useful for decoupling the frontend of the compiler from the code generator, but it is the form read by many analysis programs. Machine-independent optimizations such as common-subexpression elimination and loop-invariant removal operate on graphs expressed in IF1.

5.3 Extensions

5.3.1 IF1 Optimizer

This can be done as a immediate extension of this thesis; most of the classical code optimization techniques can be easily applied to IF1.

5.3.2 Optimizer and IF2 Code Generator

This is a machine specific optimizer. This converts the IF1 code into a machine dependent language IF2 (Intermediate Form 2). If this phase is included in the compiler then some modifications need to be made to Code Generator for Twine RISC as IF2 contains some machine specific instructions.

5.4 Improvements

5.4.1 Register Allocation

In the present implementation of the *Register Allocation* function, **usage_count** is used as one of the criteria to select the register to be deallocated. The usage_count of a variable is determined based on the number of out-going edges of that variable. But some type of nodes use the value of the variable (arithmetic nodes), some use the address of the variable (compound nodes) and others use both the address and the value of the variable (array manipulation nodes). If the **usage_count** of the variable is determined based on both the number of out-going edges and to which the edges are leading to then much more efficient allocation algorithm may be obtained.

5.4.2 Exploit TRS

Once the phases, *IF1 Optimizer* and the *Optimizer* are included in the compiler, the Twine RISC streams[RS94] can be easily exploited.

5.5 Conclusions

A Translator for IF1(Intermediate Form 1) to Twine RISC has been implemented in an novel and efficient manner. The translator can serve as a common phase in the compilers that produce Twine RISC code. The source language of this compilers may be a functional language or a data-flow language. In this implementation the size of the derived data types of IF1 can be determined at run-time which makes the language more flexible.

The code generated is not very efficient because it is not able to utilize all the Twine RISC Streams available. This is due to non-existence of the Optimizer(section 5.3.2) and the IF1 Optimizer(section 5.3.1).

Bibliography

[AC86] Arvind and D. Culler, *Dataflow Architectures*, Annual Reviews in Computer Science, Vol 1, Annual Reviews Inc., Palo Alto, CA, 1986. pp 225-253.

[AN89] Arvind, Rishiyur and S. Nikhil, *Executing a Program on the MIT Tagged-Token Dataflow Architecture*, IEEE *Trans. Comp.*. 1989.

[ASU86] Aho, Ravi Sethi and Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesely Publishing Company, 1986.

[CAO88] John P Hayes *Computer architecture and organization* , second edition , McGraw Hill, 1988. pp 194-198.

[DH92] Dhiren Patel, *Twine-RISC : Architecture and Performance Evaluation Study*, Master's Thesis, Department of Computer Science, IIT Kanpur, March 1992.

[DIN93] Dinesh Rao, *Design of Twine-RISC*, Master's Thesis, Department of Computer Science, IIT Kanpur, March 1993.

[FFM88] Programers Manual SUN i386 : File Formats , SUN Inc. 1988.

[NA89] R. Nikhil and Arvind, *Can Dataflow Subsume Von Neumann computing?*, Proc. 16^{th} Int. Symp. On Computer Architecture, Jerusalem, Israel, June 1989. pp 262-272.

[PS82] David A. Patterson and Carlo H. Sequin, *A VLSI RISC*, IEEE Computer, Sept. 1982, pp.8-21.

[RIY93] Riyaz V P, *Frontend of a SISAL compiler* , Master's Thesis, Deparament of Computer Science, IIT Kanpur, March 1993.

[RS94] Rajat Moona and Sanjeev Kumar, *Twine RISC: A high performance multi-threaded RISC architecture*, International workshop on parallel processing, Bangalore, Dec '94.

[SIS85] James McGraw, Stephen, Allan, Oldehoeft, John, Chris, Bill and Robert, SISAL (Streams and Iterations in a Single Assignment Language, Language Reference Manual Version 1.2, 1985.

[SS94] Shyam Sundar P, *Program Development Environment for Twine-RISC* , Master's Thesis, Department of computer Science, IIT Kanpur, March 1994.

Appendix A

IF1 syntax

The IF1 grammar is given below in the extended BNF. Newline denotes the new-line charecter, Literal donates any string of the printable charecter except the new-line, PosInteger denotes a positive integer, and Integer denotes a zero or a positive integer. A string of symbols enclosed by []'s denotes an optional item. An ellipsis (...) denotes the repetition of one or more times of the symbol it follows. If the ellipsis follows an optional string it represents the repetition of the string zero or more times.

```
File        ::= [line Comment Newline ... ] ...
Line        ::= C
            |   N Label TypeTableEntry
            |   E Label Destination     TypeReference
            |   L       Destination     TypeReference   "Literal"
            |   L       Destination     TypeReference   Errorvalue
            |   G                       TypeReference
            |   G                       TypeReference   "Literal"
            |   I                       TypeReference   "Literal"
            |   X                       TypeReference   "Literal"
            |   {
            |   } Label Node            Count           AssociationList

Label       ::= PosInteger
```

```
Source          ::= SourceNode      SourcePort
Destnation      ::= DestinationNode DestinationPort
SourceNode      ::= Integer
SourcePort      ::= PosInteger
DestinationNode ::= Integer
DestinationPort ::= PosInteger
TypeReference   ::= Integer
Count           ::= Integer
AssociationList ::= Integer...
Node Forall     ::= | Select | Tagcase | LoopA | LoopB | AAddH |
                    | AExtract | ABuild | ACatenate | AElement | Agather
                    | AIsEmpty | ALimH | ALimL | ARemL | ARemH | Abs
                    | AScatter | ASetL | ASize | Bool | Call | Char | Div
                    | Double | Exp | FirstValue | FinalValue | Floor | Int
                    | IsError | LessEqual | Max | Min | Minus | Mod | Neg
                    | Not | NotEqual | Plus | RangeGen | RBuild | RElements
                    | RReplace | RedLeft | RedRight | RedTree | Reduce
                    | RestValues | Times | Trunc | AAddL | Single | NoOp
                    | Less | Equal | ARepalce | AFill
TypeTableEntry ::= Array        Typereference
Basic          ::= BasicType
                 | Field
                 | Function
                 | Multiple
                 | Record
                 | Stream
                 | Tag
                 | Tuple
                 | Union
BasicType      ::= Boolean
                 | Character
                 | Double
```

| Integer

| Null

| Real

Appendix B

Twine RISC Mnemonic Table

#	Instruction	Description of the Instruction
1	add rs1, rs2, rd	Adds contents of rs1 & rs2 and stores the result in rd.
2	sub rs1, rs2, rd	Subtracts contents of rs2 from contents of rs1 and stores the result in rd.
3	and rs1, rs2, rd	Logical AND of contents of rs1 & rs2 is stored in rd.
4	or rs1, rs2, rd	Logical OR of contents of rs1 & rs2 is stored in rd.
5	xor rs1, rs2, rd	Logical XOR of contents of rs1 & rs2 is stored in rd.
6	sftl x, rs, rd	Shifts the contents of rs to the left by x-bits and stores the result in rd.
7	sftr x, rs, rd	Shifts the contents of rs to the right by x-bits and stores the result in rd.
8	mvi rd, x	Moves x to register rd.
9	jmp <label>	Unconditional jump to <label>.
10	jump rs	Unconditional jump to a location whose offset is in rs.
11	jz rs, <label>	Jump to <label> if contents of rs is zero (equal to).

Cont'd ...

#	Instruction	Description of the Instruction
12	jp rs, <label>	Jump to <label> if contents of rs is positive (greater than).
13	jpz rs, <label>	Jump to <label> if contents of rs is positive or zero (greater than or equal to).
14	jnz rs, <label>	Jump to <label> if contents of rs is negative or zero (less than or equal to).
15	mfork rs, rd	generate threads using contents of rs and store no. of threads generated in rd.
16	mjoin rs, rs	Decrement contents of rs and test it, if it is zero thread continues else thread dies.
17	chfp rs	Loads the first six bits of rs to FP.
18	load rs, rd	Load the contents of memory address [rs] into rd.
19	loadx a, rd	Load the contents memory address a into rd.
20	resm	Move data from data queue to register.
21	store rd, rs	Store the contents of rs into memory address [rd].
22	storex x, rs	Store the contents of rs into memory address x.

Appendix C

Test Cases

The below listing is the Code generated by the Code generator for some of the IF1 programs. The generated code is tested using the simulator for Twine RISC and the results were found to be correct.

IF1(Intermediate Form I) Program 1: This Program adds two numbers, subtracts one number from another and tests whether first result is less than or equal to second.

```
C "standard Types"
T 38 1       0 %na=Boolean
T 2 1        1 %na=Character
T 3 1        2 %na=Double
T 4 1        3 %na=Integer
T 5 1        4 %na=Null
T 6 1        5 %na=Real
T 9 0            4
T 1 1        0 %na=Boolean
T 10 7 1 0
T 11 7 6 10
T 12 7 9 11
T 13 7 4 12
T 14 9 13
T 15 3 15 15
T 17 4 4
T 18 4 1

G   15    testsum
E 0 1 1 1 4
E 0 2 1 2 4
N 1 Plus
E 0 3 2 1 4
E 0 2 2 2 4
N 2 Minus
E 2 1 3 1 4
E 1 1 3 2 4
N 3 LessEqual
E 3 1 0 1 1
```

```
G  15 main
L 1 1 15 testsum
L 1 4 4 2
L 1 2 4 3
L 1 3 4 -4
N 1 Call
E 1 1 0 1 1
```

Generated Twine RISC code for Program 1:

```
        MACRO  plus_int Arg_a1,Arg_a2,Arg_a3,Arg_a5
        LOCAL N1,N3
        sftl 29,r61,Arg_a3
        sub Arg_a3,Arg_a1,Arg_a3
        jz Arg_a3,N3
        sftl 29,r61,Arg_a3
        sub Arg_a3,Arg_a2,Arg_a3
        jz Arg_a3,N3
        add Arg_a1,Arg_a2,Arg_a3
        jp Arg_a1,N1
        jpz Arg_a2,Arg_a5
        jz Arg_a1,Arg_a5
        jpz Arg_a3,N3
        jmp Arg_a5
N1:     jnz Arg_a2,Arg_a5
        jp Arg_a3,Arg_a5
N3:     sftl 29,r61,Arg_a3
        MEND

        MACRO  minus_int Arg_a1,Arg_a2,Arg_a3,Arg_a5
        LOCAL N1,N3
        sftl 29,r61,Arg_a3
        sub Arg_a3,Arg_a1,Arg_a3
        jz Arg_a3,N3
        sftl 29,r61,Arg_a3
        sub Arg_a3,Arg_a2,Arg_a3
        jz Arg_a3,N3
        sub Arg_a1,Arg_a2,Arg_a3
        jp Arg_a1,N1
        jnz Arg_a2,Arg_a5
        jz Arg_a1,Arg_a5
        jpz Arg_a3,N3
        jmp Arg_a5
N1:     jpz Arg_a2,Arg_a5
        jp Arg_a3,Arg_a5
N3:     sftl 29,r61,Arg_a3
        MEND

        MACRO  lesseq_int Arg_a1,Arg_a2,Arg_a3,Arg_a4,Arg_a5
        LOCAL N2,N3,N4
        sftl 29,r61,Arg_a3
        sub Arg_a1,Arg_a3,Arg_a4
        jz Arg_a4,Arg_a5
        sub Arg_a2,Arg_a3,Arg_a4
        jz Arg_a4,Arg_a5
        sftr 2,r61,Arg_a3
        jp Arg_a1,N3
        jpz Arg_a2,Arg_a5
N4:     sub Arg_a1,Arg_a2,Arg_a4
        jp Arg_a4,N2
        jmp Arg_a5
N2:     sub Arg_a3,Arg_a3,Arg_a3
        jmp Arg_a5
```

```
mvi r0,0
sftl 12,r0,r0
mvi r0,2
sub r1,r1,r1
mvi r1,8
add r63,r1,r1
store r1,r62
sftl 0,r62,r1
store r62,r0
add r61,r62,r62
mvi r2,0
sftl 12,r2,r2
mvi r2,0
sftl 12,r2,r2
mvi r2,3
sub r3,r3,r3
mvi r3,12
add r63,r3,r3
store r3,r62
sftl 0,r62,r3
store r62,r2
add r61,r62,r62
mvi r4,255
sftl 12,r4,r4
mvi r4,4095
sftl 12,r4,r4
mvi r4,4092
sub r5,r5,r5
mvi r5,16
add r63,r5,r5
store r5,r62
sftl 0,r62,r5
store r62,r4
add r61,r62,r62
sftl 0,r63,r4
sub r0,r0,r0
mvi r0,144
add r63,r0,r63
sub r0,r0,r0
mvi r0,1
sftl 2,r0,r0
add r61,r63,r2
add r0,r2,r2
store r2,r3
add r61,r2,r2
store r2,r5
add r61,r2,r2
store r2,r1
add r61,r2,r2
sftl 0,r63,r0
mvi r0,0
sftl 12,r0,r0
mvi r0,0
sftl 12,r0,r0
mvi r0,453
store r63,r0
jmp testsum
sftl 0,r63,r0
sub r2,r2,r2
mvi r2,144
sub r63,r2,r63
add r61,r0,r2
sub r1,r1,r1
```

```
        mvi r1,4
        add r4,r1,r1
        load r2,r0
        store r1,r0
        sftl 0,r0,r1
        add r61,r2,r2
        jmp end
end:    sub r61,r61,r61
```

IF1(Intermediate Form I) Program 2 : This program tests the compound node Select.

```
C "standard Types"
T 38 1       0 %na=Boolean
T 2 1       1 %na=Character
T 3 1       2 %na=Double
T 4 1       3 %na=Integer
T 5 1       4 %na=Null
T 6 1       5 %na=Real
T 9 4            1
T 10 4           4
T 1 1       0 %na=Boolean
T 11 6        3
T 12 7       6 0
T 13 3          12 12
T 14 2            4 0
T 15 8       11 0
T 16 3           14 15


G 16 selecttest
E 0 1 1 1 4
E 0 2 1 2 4
E 0 3 1 3 4
E 0 4 1 4 4
{
G 0 Predicate
N 1 plus
E 0 1 1 1 4
E 0 2 1 2 4
N 2 Plus
E 0 3 2 1 4
E 0 4 2 2 4
N 3 Less
E 1 1 3 1 4
E 2 1 3 2 4
N 4 Int
E 3 1 4 1 1
E 4 1 0 1 4

G 0 True subgraph
E 0 1 0 1 4
E 0 2 0 2 4

G 0 False subgraph
E 0 3 0 1 4
E 0 4 0 2 4
} 1 Select 3 0 2 1
E 1 1 0 1 4
```

```
E 1 2 0 2 4

G 16 main
L 1 1 16 selecttest
L 1 2 4 5
L 1 3 4 -10
L 1 4 4 34
L 1 5 4 89
N 1 Call
E 1 1 2 1 4
E 1 2 2 2 4
N 2 Min
E 2 1 0 3 4
E 1 1 0 1 4
E 1 2 0 2 4
```

Generated Twine RISC code for Program 2 :

```
        MACRO  plus_int Arg_a1,Arg_a2,Arg_a3,Arg_a5
        LOCAL N1,N3
        sftl 29,r61,Arg_a3
        sub Arg_a3,Arg_a1,Arg_a3
        jz Arg_a3,N3
        sftl 29,r61,Arg_a3
        sub Arg_a3,Arg_a2,Arg_a3
        jz Arg_a3,N3
        add Arg_a1,Arg_a2,Arg_a3
        jp Arg_a1,N1
        jpz Arg_a2,Arg_a5
        jz Arg_a1,Arg_a5
        jpz Arg_a3,N3
        jmp Arg_a5
N1:     jnz Arg_a2,Arg_a5
        jp Arg_a3,Arg_a5
N3:     sftl 29,r61,Arg_a3
        MEND
        MACRO  min_int Arg_a1,Arg_a2,Arg_a3,Arg_a4,Arg_a5
        LOCAL N2,N3,N4
        sftl 29,r61,Arg_a3
        sub Arg_a1,Arg_a3,Arg_a4
        jz Arg_a4,Arg_a5
        sub Arg_a2,Arg_a3,Arg_a4
        jz Arg_a4,Arg_a5
        sftr 0,Arg_a1,Arg_a3
        jp Arg_a1,N3
        jpz Arg_a2,Arg_a5
N4:     sub Arg_a1,Arg_a2,Arg_a4
        jp Arg_a4,N2
        jmp Arg_a5
N2:     sftr 0,Arg_a2,Arg_a3
        jmp Arg_a5
N3:     jp Arg_a2,N4
        sftr 0,Arg_a2,Arg_a3
        MEND
        MACRO  less_int Arg_a1,Arg_a2,Arg_a3,Arg_a4,Arg_a5
        LOCAL N2,N3,N4
        sftl 29,r61,Arg_a3
        sub Arg_a1,Arg_a3,Arg_a4
        jz Arg_a4,Arg_a5
        sub Arg_a2,Arg_a3,Arg_a4
        jz Arg_a4,Arg_a5
        sftr 2,r61,Arg_a3
```

```
        jp Arg_a1,N3
        jp Arg_a2,Arg_a5
N4:     sub Arg_a1,Arg_a2,Arg_a4
        jpz Arg_a4,N2
        jmp Arg_a5
N2:     sub Arg_a3,Arg_a3,Arg_a3
        jmp Arg_a5
N3:     jp Arg_a2,N4
        sub Arg_a3,Arg_a3,Arg_a3
        MEND
        MACRO  bool_int Arg_a1,Arg_a2,Arg_a3,Arg_a4
        sftl 0,Arg_a1,Arg_a2
        jz Arg_a1,Arg_a4
        sftr 2,r61,Arg_a3
        sub Arg_a3,Arg_a1,Arg_a3
        jz Arg_a3,Arg_a4
        sftl 29,r61,Arg_a2
        MEND
entry:
        mvi r63,0
        sftl 12,r63,r63
        mvi r63,0
        sftl 12,r63,r63
        mvi r63,1068
        sftl 2,r63,r62
        sub r61,r61,r61
        mvi r61,4
        jmp main
selecttest:
        sub r1,r1,r1
        mvi r1,12
        add r63,r1,r1
        load r1,r1
        load r1,r0
        sub r2,r2,r2
        mvi r2,16
        add r63,r2,r2
        load r2,r2
        load r2,r1
        plus_int r0,r1,r3,N131
N131:
        sub r2,r2,r2
        mvi r2,32
        add r2,r63,r2
        store r2,r62
        sftl 0,r62,r2
        store r62,r3
        add r61,r62,r62
        sub r1,r1,r1
        mvi r1,20
        add r63,r1,r1
        load r1,r1
        load r1,r0
        sub r4,r4,r4
        mvi r4,24
        add r63,r4,r4
        load r4,r4
        load r4,r1
        plus_int r0,r1,r5,N161
N161:
        sub r4,r4,r4
        mvi r4,36
        add r4,r63,r4
```

```
        store r4,r62
        sftl 0,r62,r4
        store r62,r5
        add r61,r62,r62
        less_int r3,r5,r1,r0,N191
N191:
        sub r0,r0,r0
        mvi r0,40
        add r0,r63,r0
        store r0,r62
        sftl 0,r62,r0
        store r62,r1
        add r61,r62,r62
        bool_int r1,r2,r0,N221
N221:
        sub r0,r0,r0
        mvi r0,28
        add r0,r63,r0
        store r0,r62
        sftl 0,r62,r0
        store r62,r2
        add r61,r62,r62
        jz r2,N101
        sub r0,r0,r0
        mvi r0,12
        add r63,r0,r0
        load r0,r0
        sub r1,r1,r1
        mvi r1,4
        add r63,r1,r1
        store r1,r0
        sftl 0,r0,r1
        sub r0,r0,r0
        mvi r0,16
        add r63,r0,r0
        load r0,r0
        sub r1,r1,r1
        mvi r1,8
        add r63,r1,r1
        store r1,r0
        sftl 0,r0,r1
        jmp N102
N101:
        sub r0,r0,r0
        mvi r0,4
        add r63,r0,r0
        store r0,r0
        sftl 0,r0,r0
        sub r0,r0,r0
        mvi r0,8
        add r63,r0,r0
        store r0,r0
        sftl 0,r0,r0
N102:
        load r63,r0
        jump r0
main:
        mvi r0,0
        sftl 12,r0,r0
        mvi r0,0
        sftl 12,r0,r0
        mvi r0,5
        sub r1,r1,r1
```

```
mvi r1,16
add r63,r1,r1
store r1,r62
sftl 0,r62,r1
store r62,r0
add r61,r62,r62
mvi r2,255
sftl 12,r2,r2
mvi r2,4095
sftl 12,r2,r2
mvi r2,4086
sub r3,r3,r3
mvi r3,20
add r63,r3,r3
store r3,r62
sftl 0,r62,r3
store r62,r2
add r61,r62,r62
mvi r4,0
sftl 12,r4,r4
mvi r4,0
sftl 12,r4,r4
mvi r4,34
sub r5,r5,r5
mvi r5,24
add r63,r5,r5
store r5,r62
sftl 0,r62,r5
store r62,r4
add r61,r62,r62
mvi r6,0
sftl 12,r6,r6
mvi r6,0
sftl 12,r6,r6
mvi r6,89
sub r7,r7,r7
mvi r7,28
add r63,r7,r7
store r7,r62
sftl 0,r62,r7
store r62,r6
add r61,r62,r62
sftl 0,r63,r6
sub r2,r2,r2
mvi r2,164
add r63,r2,r63
sub r2,r2,r2
mvi r2,2
sftl 2,r2,r2
add r61,r63,r4
add r2,r4,r4
store r4,r1
add r61,r4,r4
store r4,r3
add r61,r4,r4
store r4,r5
add r61,r4,r4
store r4,r7
add r61,r4,r4
sftl 0,r63,r2
mvi r2,0
sftl 12,r2,r2
mvi r2,0
```

```
        sftl 12,r2,r2
        mvi r2,639
        store r63,r2
        jmp selecttest
        sftl 0,r63,r2
        sub r4,r4,r4
        mvi r4,164
        sub r63,r4,r63
        add r61,r2,r4
        sub r3,r3,r3
        mvi r3,32
        add r6,r3,r3
        load r4,r2
        store r3,r2
        sftl 0,r2,r3
        add r61,r4,r4
        sub r5,r5,r5
        mvi r5,36
        add r6,r5,r5
        load r4,r2
        store r5,r2
        sftl 0,r2,r5
        add r61,r4,r4
        load r3,r2
        load r5,r3
        min_int r2,r3,r5,r4,N281
N281:
        sub r4,r4,r4
        mvi r4,12
        add r4,r63,r4
        store r4,r62
        sftl 0,r62,r4
        store r62,r5
        add r61,r62,r62
        jmp end
end:    sub r61,r61,r61
```

IF1(Intermediate Form I) Program 3: This program builds an array of integers and extracts 2nd and 3rd elements and adds these elements.

```
C "standard Types"
T 4 1       3 %na=Integer
T 9 0             4
T 1 1       0 %na=Boolean
T 10 7 1 0
T 11 7 6 10
T 12 7 9 11
T 13 7 4 12
T 14 9 13
T 15 3 15 15

G  15   arraytest
E 0 1 1 1 4
E 0 2 1 2 4
E 0 3 1 3 4
E 0 4 1 4 4
E 0 5 1 5 4
```

```
N 1 ABuild
N 2 AElement
N 3 AElement
N 4 Plus
E 1 1 2 1 9
E 1 1 3 1 9
L 2 2 4 4
L 3 2 4 3
E 2 1 4 1 4
E 3 1 4 2 4
E 4 1 0 1 4

G   15 main
L 1 1 15 arraytest
L 1 2 4 2
L 1 3 4 -5
L 1 4 4 92
L 1 5 4 -54
L 1 6 4 45
N 1 Call
E 1 1 0 1 4
```

Generated Twine RISC code for Program 3:

```
      MACRO  plus_int Arg_a1,Arg_a2,Arg_a3,Arg_a5
      LOCAL N1,N3
      sftl 29,r61,Arg_a3
      sub Arg_a3,Arg_a1,Arg_a3
      jz Arg_a3,N3
      sftl 29,r61,Arg_a3
      sub Arg_a3,Arg_a2,Arg_a3
      jz Arg_a3,N3
      add Arg_a1,Arg_a2,Arg_a3
      jp Arg_a1,N1
      jpz Arg_a2,Arg_a5
      jz Arg_a1,Arg_a5
      jpz Arg_a3,N3
      jmp Arg_a5
N1:   jnz Arg_a2,Arg_a5
      jp Arg_a3,Arg_a5
N3:   sftl 29,r61,Arg_a3
      MEND
entry:
      mvi r63,0
      sftl 12,r63,r63
      mvi r63,0
      sftl 12,r63,r63
      mvi r63,1096
      sftl 2,r63,r62
      sub r61,r61,r61
      mvi r61,4
      jmp main
arraytest:
      sub r1,r1,r1
      mvi r1,8
      add r63,r1,r1
      load r1,r1
      load r1,r0
      sub r3,r3,r3
      mvi r3,28
      add r3,r63,r3
      store r3,r62
```

```
sftl 0,r62,r3
sub r1,r1,r1
store r62,r1
add r61,r62,r62
store r62,r0
add r61,r62,r62
sub r2,r2,r2
mvi r2,4
store r62,r2
add r61,r62,r62
sub r5,r5,r5
mvi r5,12
add r63,r5,r5
load r5,r5
store r62,r5
add r61,r62,r62
sub r5,r5,r5
mvi r5,16
add r63,r5,r5
load r5,r5
store r62,r5
add r61,r62,r62
sub r5,r5,r5
mvi r5,20
add r63,r5,r5
load r5,r5
store r62,r5
add r61,r62,r62
sub r5,r5,r5
mvi r5,24
add r63,r5,r5
load r5,r5
store r62,r5
add r61,r62,r62
mvi r0,0
sftl 12,r0,r0
mvi r0,0
sftl 12,r0,r0
mvi r0,4
sub r4,r4,r4
mvi r4,32
add r63,r4,r4
store r4,r62
sftl 0,r62,r4
store r62,r0
add r61,r62,r62
sub r7,r7,r7
mvi r7,36
add r7,r63,r7
sftl 29,r61,r4
sub r4,r1,r5
jz r5,N132
sub r4,r0,r5
jz r5,N132
sub r4,r2,r5
jz r5,N132
add r61,r3,r5
load r5,r5
sub r0,r5,r5
sftr 2,r61,r6
add r6,r5,r5
jnz r5,N132
sub r5,r2,r6
```

```
        jp r6,N132
        sftl 1,r61,r4
        add r4,r3,r4
        sftl 2,r5,r5
        add r5,r4,r4
        load r4,r5
        store r7,r5
        sftl 0,r5,r7
        load r7,r4
        jmp N131
N132:
        store r7,r62
        sftl 0,r62,r7
        sftl 29,r61,r4
        store r62,r4
        add r61,r62,r62
N131:
        mvi r0,0
        sftl 12,r0,r0
        mvi r0,0
        sftl 12,r0,r0
        mvi r0,3
        sub r5,r5,r5
        mvi r5,40
        add r63,r5,r5
        store r5,r62
        sftl 0,r62,r5
        store r62,r0
        add r61,r62,r62
        sub r9,r9,r9
        mvi r9,44
        add r9,r63,r9
        sftl 29,r61,r5
        sub r5,r1,r6
        jz r6,N162
        sub r5,r0,r6
        jz r6,N162
        sub r5,r2,r6
        jz r6,N162
        add r61,r3,r6
        load r6,r6
        sub r0,r6,r6
        sftr 2,r61,r8
        add r8,r6,r6
        jnz r6,N162
        sub r6,r2,r8
        jp r8,N162
        sftl 1,r61,r5
        add r5,r3,r5
        sftl 2,r6,r6
        add r6,r5,r5
        load r5,r6
        store r9,r6
        sftl 0,r6,r9
        load r9,r5
        jmp N161
N162:
        store r9,r62
        sftl 0,r62,r9
        sftl 29,r61,r5
        store r62,r5
        add r61,r62,r62
N161:
```

```
        plus_int r4,r5,r1,N191
N191:
        sub r0,r0,r0
        mvi r0,4
        add r0,r63,r0
        store r0,r62
        sftl 0,r62,r0
        store r62,r1
        add r61,r62,r62
        load r63,r0
        jump r0
main:
        mvi r0,0
        sftl 12,r0,r0
        mvi r0,0
        sftl 12,r0,r0
        mvi r0,2
        sub r1,r1,r1
        mvi r1,8
        add r63,r1,r1
        store r1,r62
        sftl 0,r62,r1
        store r62,r0
        add r61,r62,r62
        mvi r2,255
        sftl 12,r2,r2
        mvi r2,4095
        sftl 12,r2,r2
        mvi r2,4091
        sub r3,r3,r3
        mvi r3,12
        add r63,r3,r3
        store r3,r62
        sftl 0,r62,r3
        store r62,r2
        add r61,r62,r62
        mvi r4,0
        sftl 12,r4,r4
        mvi r4,0
        sftl 12,r4,r4
        mvi r4,92
        sub r5,r5,r5
        mvi r5,16
        add r63,r5,r5
        store r5,r62
        sftl 0,r62,r5
        store r62,r4
        add r61,r62,r62
        mvi r6,255
        sftl 12,r6,r6
        mvi r6,4095
        sftl 12,r6,r6
        mvi r6,4042
        sub r7,r7,r7
        mvi r7,20
        add r63,r7,r7
        store r7,r62
        sftl 0,r62,r7
        store r62,r6
        add r61,r62,r62
        mvi r8,0
        sftl 12,r8,r8
        mvi r8,0
```

```
        sftl 12,r8,r8
        mvi r8,45
        sub r9,r9,r9
        mvi r9,24
        add r63,r9,r9
        store r9,r62
        sftl 0,r62,r9
        store r62,r8
        add r61,r62,r62
        sftl 0,r63,r4
        sub r0,r0,r0
        mvi r0,152
        add r63,r0,r63
        sub r0,r0,r0
        mvi r0,1
        sftl 2,r0,r0
        add r61,r63,r2
        add r0,r2,r2
        store r2,r1
        add r61,r2,r2
        store r2,r3
        add r61,r2,r2
        store r2,r5
        add r61,r2,r2
        store r2,r7
        add r61,r2,r2
        store r2,r9
        add r61,r2,r2
        sftl 0,r63,r0
        mvi r0,0
        sftl 12,r0,r0
        mvi r0,0
        sftl 12,r0,r0
        mvi r0,753
        store r63,r0
        jmp arraytest
        sftl 0,r63,r0
        sub r2,r2,r2
        mvi r2,152
        sub r63,r2,r63
        add r61,r0,r2
        sub r1,r1,r1
        mvi r1,4
        add r4,r1,r1
        load r2,r0
        store r1,r0
        sftl 0,r0,r1
        add r61,r2,r2
        jmp end
end:    sub r61,r61,r61
```